TAITA TAVETA UNIVERSITY

**HBT: 2403**

SYSTEM PROJECT; SUBMISSION OF PROJECT REPORT

**GROUP 2**

**NAME REG. NO**

Dennis Mutheke TU01-BE213-0375/2017

Stephen N. Kibe TU01-BE213-0398/2017

Maria Mwangiri TU01-BE213-0378/2017

Melvin M. Mmbuka TU01-BE213-0386/2017

Sheldon Mambo TU01-BE213-0373/2017

Brian Rotich TU01-BE213-0406/2016

**AUTOMATIC IMAGE CLASSIFICATION FOR LOST DOCUMENTS IN KENYA**

**Step 1: Importing Required Libraries**

In this phase, we will make use of the TensorFlow and Keras libraries for creating our model and training it. In addition to, we will also use Matplotlib for visualizing our dataset to gain a better understanding of the images we are going to handle. TensorFlow is an open-source library used for building models which would result in the tasks like image recognition/classification. Keras is in high level application programming interface that can use TensorFlow’s functions to provide for loading, converting and saving image data. NumPy is a library for the Python programming language used for adding support in large multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from keras.preprocessing import image

from keras.models import load\_model

from tensorflow.keras.optimizers import RMSprop

import tensorflow as tf

from matplotlib import pyplot as plt

import cv2

import os

import numpy as np

import tensorflow as tf

#### Step 2: Prepare Dataset for Training

This dataset contains images of documents with accompanying document type labels. There are three different types of documents grouped into three different folders (according to document type). The documents types are Identification cards, Certificates and Title deeds. The dataset is divided into Training data, Validation data and Testing data.

After splitting the dataset for training, we will involve assigning paths and creating categories (labels), resizing our image into 100,100. Next, we need to reshape our dataset inputs to the shape that our model expects when we train the model. Since 255 is the maximum pixel value, rescale 1/255 is used to transform every pixel value in the range of (0,255) -(0,1).

The ImageDataGenerator function reads the images from a big NumPy array and folders containing images.

In [2]: train = ImageDataGenerator(rescale=1/255)

validation = ImageDataGenerator(rescale=1/255)

The train. flow\_from\_directory() is a method of ImageDataGenerator class that reads the images from a big NumPy array.

The target size is the size of our input images, every image will be resized to this size.

The batch size is the number to be yielded from the generator per batch.

We used the class mode categorical because we have more than two classes that is certificates, IDs and title deeds.

In [3]: train\_dataset = train. flow\_from\_directory(r"S:\Documents\train",

target\_size= (100,100),

batch\_size= 10,

class\_mode= "categorical")

validation\_dataset = train.flow\_from\_directory(r"S:\Documents\validation",

target\_size= (100,100),

batch\_size= 10,

class\_mode= "categorical")

Out [3]: Found 200 images belonging to 3 classes.

Found 140 images belonging to 3 classes.

We need to one-hot-encode our target variable this means that a column will be created for each output category and a binary variable is inputted for each category. To do this we convert the label into a set of 3 numbers where each number represents a class. So, if an image belongs to the first class, the first number of this set will be a 1 and all other numbers of this set will be 0.

In [4]: train\_dataset.class\_indices

Out [4]: {'CERTIFICATES': 0, 'IDENTIFICATION CARDS': 1, 'TITLE DEEDS': 2}

#### Step 3: Build the Model

#### First, we need to create the convolutional layers. You apply different filters to allow the network to learn important feature. You specify the size of the kernel and the number of filters.

The architecture of our CNN model;

* Conv2D layer – we will add 3 convolutional layers of 3\*3 filter size, and activation function as ReLu.
* Max Pooling – MaxPool2D with 2\*2 polling size and stride 2.
* 2nd Convolutional layer of 3\*3 filter size and stride 2
* Max Pooling – MaxPool2D with 2\*2 polling size and stride 2.
* 3nd Convolutional layer of 3\*3 filter size and stride 2
* Max Pooling – MaxPool2D with 2\*2 polling size and stride 2.
* One Flatten layer
* Dense, feed-forward neural network (activation =” ReLu”)
* Two Dense layer (activation =” softmax”)

MaxPool2D – Maximum pooling layer is used to reduce the size of images

Flatten – Flattening operation is converting the dataset into a 1-D array for input into the next layer which is the fully connected layer.

Dense layer – for feed-forward neural network and helps to decide which category an image belongs.

The last layer will have an activation function as softmax for multi-class classification.

In [5]: model **=** tf**.**keras**.**models**.**Sequential([ tf**.**keras**.**layers**.**Conv2D(16,(3,3),activation **=** 'relu tf**.**keras**.**layers**.**MaxPool2D(2,2),

tf**.**keras**.**layers**.**Conv2D(32,(3,3),activation **=** 'relu' tf**.**keras**.**layers**.**MaxPool2D(2,2),

tf**.**keras**.**layers**.**Conv2D(64,(3,3),activation **=** 'relu' tf**.**keras**.**layers**.**MaxPool2D(2,2),

tf**.**keras**.**layers**.**Flatten(),

tf**.**keras**.**layers**.**Dense(128,activation**=** 'relu'),

tf**.**keras**.**layers**.**Dense(3,activation**=** 'softmax') ])

**Step 4: Compile and Train the Model**

Before the training process starts, we need to configure the learning process, which is done via the compile method. It receives three arguments:

A loss function which is the objective that the model will try to minimize. We use a string identifier of an existing loss function which is categorical\_crossntropy.

An optimizer which is our string identifier which is RMSprop and a learning rate of 0.001.

Metrics for any classification problem which we set to ‘accuracy’.

In [6]: model.compile(loss= 'categorical\_crossentropy', optimizer = RMSprop(lr=0.001), metrics =['accuracy'])

For training we use the model.fit. Epoch is the number of times the model will be trained. We used 10 epochs and 20 step\_per\_epoch which define the total number of steps(batches of samples) to yield the generator before declaring one epoch finished and starting the next epoch.

In [7]: model\_fit **=** model**.**fit(train\_dataset, step\_per\_epoch **=** 20, epochs **=** 10, validation\_data **=** validation\_dataset)

Out [7]:

Epoch 3/1014/14 [==============================] - 193s 14s/step - loss: 0.5089 - acc: 0.835720/20 [==============================] - 298s 15s/step - loss: 0.6425 - acc: 0.7500 - va l\_loss: 0.5089 - val\_acc: 0.8357Epoch 4/1014/14 [==============================] - 211s 15s/step - loss: 0.4745 - acc: 0.850020/20 [==============================] - 312s 16s/step - loss: 0.5745 - acc: 0.7700 - va l\_loss: 0.4745 - val\_acc: 0.8500Epoch 5/1014/14 [==============================] - 214s 15s/step - loss: 0.3988 - acc: 0.878620/20 [==============================] - 323s 16s/step - loss: 0.4097 - acc: 0.8450 - va l\_loss: 0.3988 - val\_acc: 0.8786Epoch 6/1014/14 [==============================] - 207s 15s/step - loss: 0.5645 - acc: 0.814320/20 [==============================] - 313s 16s/step - loss: 0.4302 - acc: 0.8350 - va l\_loss: 0.5645 - val\_acc: 0.8143Epoch 7/1014/14 [==============================] - 133s 10s/step - loss: 0.4455 - acc: 0.878620/20 [==============================] - 223s 11s/step - loss: 0.2577 - acc: 0.9150 - va l\_loss: 0.4455 - val\_acc: 0.8786Epoch 8/1014/14 [==============================] - 132s 9s/step - loss: 0.5718 - acc: 0.857120/20 [==============================] - 221s 11s/step - loss: 0.1978 - acc: 0.9350 - va l\_loss: 0.5718 - val\_acc: 0.8571Epoch 9/1014/14 [==============================] - 130s 9s/step - loss: 0.2347 - acc: 0.942920/20 [==============================] - 208s 10s/step - loss: 0.2816 - acc: 0.8900 - va l\_loss: 0.2347 - val\_acc: 0.9429Epoch 10/1014/14 [==============================] - 138s 10s/step - loss: 0.2475 - acc: 0.964320/20 [==============================] - 215s 11s/step - loss: 0.1263 - acc: 0.9750 - val\_loss: 0.2475 - val\_acc: 0.9643

After training we achieved an accuracy of 96% and saved our model as doclassification.h5 using model.save.

In [8]: model.save('doclassification.h5')

## **Step 6: Make Prediction**

First, we import our model using the load\_model function. To see predictions that our model has made for the test data, we use the predict function. The predict function will give an array with 3 numbers. These numbers are the probabilities that the input image represents each digit (0–2). If the number is 0 the image will be classified as a CERTIFICATE, 1 for IDENTIFICATION CARD and 2 for TITLE DEEDS. We used cv2 to load a sample certificate image from our documents folder, then resized it to 100 by 100 and used NumPy to reshape the image so as to feed it into the model. The predict function predicts the image as a certificate

In [9]: model = load\_model('doclassification.h5')

In [10]: img = cv2.imread(r"S:\Documents\test\img1.jpg")

img = cv2.resize(img, (100,100))

img = np.reshape(img,[1,100,100,3])

In [12]: classes = model.predict\_classes(img)

if classes == 0:

print ("0: CERTIFICATE")

elif classes == 1:

print ("1: IDENTIFICATION CARD")

else:

print ("2: TITLE DEED")

Out [13]: 0: CERTIFICATE